As usual throughout the volume, anything which might be construed as a proof is tucked away in small print as a double starred section which can easily be omitted without too great a feeling of guilt. Various sorting algorithms are compared to illustrate time efficiency and the analysis of complexity. Upper and lower bounds on the complexity of some algorithms are obtained as a prelude to the usual classification study.

Limitations and Robustness are the subject of the third part. The distinction between polynomial and exponential time algorithms is nicely put by referring to them as being "reasonable" or "unreasonable". This leads naturally to a discussion of the class of NP complete problems, the hardest problems for which a "certificate" can be given which allows the answer to be checked in polynomial time. Beyond this class, there are progressively harder problems, crowned by those for which no algorithm exists. These are illustrated by tiling problems, where one has to try to cover a plane or half-plane with square tiles with coloured edges, so that adjacent edges have the same colour. Perhaps more useful, a program verifier is shown not to be too lovable! The non-existence of a nice one is illustrated by the problem of establishing the termination of a program. It is easier to consider this halting problem for a much simpler machine, the Turing machine, which is polynomially equivalent to the most powerful of today's machines (and all those that will ever be built, according to Church's thesis).

The final part treats much more recent work which has broadened the whole area of algorithms. The main topics are parallelism and concurrency, probabilistic algorithms, and artificial intelligence. The first is right up-to-date with mention of an algorithm for sorting N numbers using N processors in log N time. The second is illustrated with the RSA public key cryptosystem, and the last introduces the notion of heuristics for pruning search spaces. One of the best of the Old Testament quotations which introduce and conclude each chapter, was one from Ezekiel for this chapter on intelligent machines, "Son of man, can these bones live?".

The excellent bibliographic notes are both comprehensive and graded. The standard might be judged from the knowledge that the author refers to a journal called the "Science of Computer Programming" (p. 365) and, under the topic of recursion, heartily recommends, as this author does, a book by D. Harel, "Algorithmics—The Spirit of Computing".

C. D. WALTER
UMIST, Manchester
United Kingdom

The Little LISPer. By Daniel P. Friedman and Matthias Felleisen. MIT Press, Cambridge, MA, trade ed., 1987, Price £9.95.

The title, *The Little LISPer*, is somewhat misleading: the authors' avowed purpose in writing this book is not so much to teach LISP, as to teach *recursive thinking.*

Programming languages, they explain in the preface, are good languages for teaching recursion, and of programming languages, the best for this purpose is LISP.

LISP, then, is the medium, recursion the message. Accordingly, only a small subset of LISP is covered—no more than the authors consider necessary to present an edifying variety of recursive function definitions. Much of the book (the first eight chapters) is taken up with leading the reader through definitions of what, in most programming texts, is taken for granted: most notably, the definitions of addition, subtraction, multiplication and exponentiation (on the natural numbers) in terms of the primitives add1 and sub1. Other, more familiar examples of recursive programs involving list- and set-manipulation are also explained in detail. The exposition is gentle, non-mathematical and should be thoroughly accessible to a reader with no background in recursion.

The last two chapters, by contrast, demand a greater degree of concentration (and, perhaps, of intellectual sophistication). Chapter 9 explains how lambda-expressions enable one to dispense with function-names, even of recursively defined functions; chapter 10 takes the reader through the construction of a sample LISP interpreter (written in LISP). The elegance and simplicity of these final chapters is impressive by any standards, so much so that they can be a pleasure even for an experienced programmer.

Style and presentation are light-hearted, and a little unusual. The entire text is a series of questions and answers, interspersed with periodic prandial exhortations (I think that not a page goes by without a (sometimes sickening) suggestion about what to eat next) and with cartoon drawings of baby elephants engaging in sundry enjoyable activities. Some (though not I) may be irritated by these irrelevancies; others (though again, not I) may find them charming. But whatever one's attitude to the authors' stylistic decisions, one cannot fail to admire the care and attention with which those decisions are implemented. The layout is spacious and uncluttered, the typography and punctuation consistent and helpful, the style unpretentious and graceful.

One disturbing feature of this book is illustrated by the following excerpt (p. 6):

> Q: What is (car *l*) where *l* is the argument (((**hotdogs**)) (**and**) (**pickle**) **relish**)
>
> A: ((**hotdogs**)), because (car *l*) is another way to ask for "the car of the list *l*."

The trouble lies in the (rigorously consistent) use of the verb *to be* to denote the relationship between a LISP expression and its value. Imagine a novice LISPer reading the passage just quoted. So he sits down at a terminal, types (car (((hotdogs)) (and) (pickle) relish)), and gets—an error message! And there is nothing in the book for quite a while to explain why. The novice cannot be expected to know that when the book says "*l* is (((**hotdogs**)) (**and**) (**pickle**) **relish**)", what that means is "*l has the value* (((**hotdogs**)) (**and**) (**pickle**) **relish**)." Nor can the novice be expected to know that LISP evaluates a function call by first evaluating the arguments and then applying the function, for this fact is not properly explained early in the book. True,

the fact that *l* is carefully printed in italics indicates that it is a *parameter*, but the reader at this stage cannot reasonably be presumed to appreciate the implications of this. True also, the example of (car (car *l*)), which immediately follows the passage just quoted, illustrates the point that inner function calls are evaluated before outer ones; but the reader is given no hint that non-numeric atoms can be given values in LISP (except as function parameters) and will thus have difficulty experimenting for himself with the material presented. (**quote** is not explained until p. 107; **setq** is not mentioned at all.)

This approach to (or, should I say, *avoidance of*) LISP evaluation, a subject which other books on LISP treat as central and elementary, is both consistent and—I have no doubt—deliberate. But that by itself does not mean that it is a good thing. For the novice programmer needs another book on LISP (or, alternatively, a teacher), not when he has *finished The Little LISPer*, but when he is *starting*. Conceptually elegant and economical though it is, *The Little LISPer* makes too few concessions to the practicalities of learning programming to be useful on its own, *even as a start*. And it is at this point that the central problem arises: almost any book explaining these missing practicalities is certain to cover much of the material in the first eight chapters of *The Little LISPer* anyway, thus resulting in a needless duplication.

The average newcomer to LISP needs an explanation not only of **setq** and the LISP evaluation process, but also of such staples as property lists, strings, floating-point numbers, destructive list-manipulation, and iterative constructs like **prog** and **do**. *The Little LISPer* offers none of these things. If such things are the bread and butter of LISP programming, then the authors' attitude is: "Let them eat cakes".

These shortcomings apart, however, *The Little LISPer* is in many ways an admirable book, for the reasons I have already given. In particular, for a student who finds recursive thinking baffling and unintuitive—alas, there are many such—and who has access to a book (or a teacher) to explain some of the more practical aspects of interacting with a LISP interpreter, here is an enlightened, entertaining and yet informed approach, which will anticipate and mitigate many of the problems he is likely to encounter.

Ian Pratt
Department of Computer Science
Manchester University
Manchester, United Kingdom

A Review of Ada Tasking. By Alan Burns, Andrew M. Lister and Andrew J. Wellings. Lecture Notes in Computer Science 262, Springer, Berlin, 1987, 141 pp., Price £10.00 (soft cover), ISBN 3 540 18008 8.

The aim of this book is to draw on the available literature to present a comprehensive review of the Ada tasking model. The model is examined in terms of its treatment